TD #0 — Comparaisons (avec solutions)

Deux exercices classiques des livres & polycopiés d'algorithmique, presque sans prérequis !

Exercice 1. *Histoire de boulons*

Dans une boite à outils, vous disposez de n écrous de diamètres tous différents et des n boulons correspondants. Malheureusement ceux-ci sont *en vrac*, et vous voulez apparier chaque écrou avec le boulon qui lui correspond. Les différences de diamètre entre les écrous sont tellement minimes qu'il n'est pas possible de déterminer à l'œil nu si un écrou est plus grand qu'un autre, et de même pour les boulons. Par conséquent, le seul type d'opération autorisé consiste à essayer un écrou avec un boulon, ce qui peut amener trois réponses possibles : soit l'écrou est strictement plus large que le boulon, soit il est strictement moins large, soit ils ont exactement le même diamètre (youpi !).

1. Proposez un algorithme (décrit informellement mais précisément) qui apparie chaque boulon avec son écrou, et effectue au plus n^2 comparaisons boulon-écrou dans le *pire cas*.

   On choisit un premier boulon arbitraire et on le compare à tous les écrous possibles jusqu'à trouver celui qui lui correspond, ce qui prend au plus $n-1$ comparaisons (car on sait qu'un écrou correspond *forcément*). Il reste $n-1$ boulons & écrous à apparier, pour lesquels on applique le même processus. Le nombre de comparaisons dans le pire cas est alors $(n-1)+(n-2)+\cdots+1$, soit $\sum_{i=1}^{n-1} i = n(n-1)/2 \leq n^2$.

On cherche maintenant uniquement à trouver le plus petit boulon et l'écrou correspondant.

2. Proposez un algorithme (décrit informellement) qui résout ce problème en au plus $2n-2$ comparaisons dans le pire cas.

   On choisit un premier boulon arbitraire et on le compare à tous les écrous possibles (que l'on collecte en un petit tas, où l'on distingue éventuellement l'écrou correspondant au boulon au cas où on le trouverait) jusqu'à en trouver un (appelons le E_1) qui est *trop petit* (si cela n'arrive jamais, on a par chance déjà trouvé le plus petit boulon et son écrou). On compare maintenant tous les boulons restants (que l'on collecte en un petit tas, où l'on distingue etc.) à E_1 jusqu'à en trouver un (appelons le B_1) qui est trop petit pour E_1 (si cela n'arrive jamais, on a par chance déjà trouvé le plus petit écrou et son boulon). On continue ainsi de suite en comparant B_1 aux écrous restants jusqu'à en trouver un trop petit, puis cet écrou à tous les boulons restants jusqu'à en trouver un trop petit, etc.
   Pour montrer que cet algorithme satisfait le coût cible, le plus malin (?) est d'analyser celui-ci globalement en constatant qu'il y a $2n$ boulons & écrous, qu'il est suffisant pour résoudre le problème d'éliminer $2n-2$ boulons & écrous comme étant trop grands, et que dans l'algorithme décrit ci dessus, chaque « gabarit » éliminé permet d'éliminer avec lui tous les écrous ou boulons collectés en tas, qui sont en nombre égal au nombre de comparaisons effectuées avec le gabarit moins une (celle où c'est le gabarit lui-même qui est éliminé) ; autrement dit, chaque gabarit éliminé a été utilisé pour k comparaisons et permet d'éliminer k boulons ou écrous. On conclut en observant qu'un gabarit non éliminé fait forcément partie du résultat et a permis de trouver son boulon/écrou associé en un nombre de comparaisons égal au plus au nombre d'écrous (si c'est un boulon) restants au moment où le gabarit commence à être utilisé. Soit $n-x$ ce nombre et $n-y$ le nombre restant de l'autre objet (obtenus par ce qui précède après $x+y$ comparaisons avec $x < n-1$, $y < n-1$), la solution au problème a donc été trouvée avec $x+y+(n-x) = n+y < 2n-1$ comparaisons.

Exercice 2. *Min/max simultané*

On considère un ensemble $\mathcal{S}$ de n entiers, et l'on s'intéresse au problème d'en trouver l'élément maximum et l'élément minimum. La seule opération autorisée sur les éléments de l'ensemble est la comparaison entre deux d'entre eux. Il n'est par exemple pas possible de les comparer à d'autres entiers, de les utiliser dans des expressions arithmétiques, etc. Cette opération est également la seule que l'on comptera pour mesurer le *coût* d'un algorithme, et l'on s'intéressera également uniquement au coût *pire cas*.

1. Proposez un algorithme (décrit informellement mais précisément) qui résout naïvement ce problème.

   On recherche séparément le minimum et le maximum en utilisant à chaque fois l'algorithme immédiat. On décrit ici la version « minimum » par le pseudocode Python suivant :

```python
def min(S):
    m = S.get_element() # un élément arbitraire de S
    for x in S.difference(m): # (différence ensembliste)
        if x < m:
            m = x
    return m
```

REMARQUE : l'énoncé demandait une description *informelle* de l'algorithme, mais il était ici sans doute plus simple & rapide d'en donner une description formelle. Il est sans doute raisonnable de prendre une telle liberté *dans ce sens*, c'est à dire de fournir une réponse plus formalisée que ce qui est demandé, tant que l'usage de pseudocode ou d'un langage de programmation n'est pas explicitement interdite.

2. Combien de comparaisons (en fonction de n) sont-elles effectuées dans le pire cas par cet algorithme ?

$2n-2$, si l'on évite de comparer un élément à lui-même (immédiat).

On souhaite développer un algorithme plus efficace. Une idée pour cela est de commencer par regrouper les éléments de l'ensemble par paires ordonnées.

3. Proposez un algorithme (décrit informellement mais précisément) qui utilise cette approche pour résoudre le problème.

On forme $\lfloor n/2 \rfloor$ paires arbitraires d'éléments, que l'on ordonne avec autant de comparaisons. On sait alors que l'élément maximum (resp. minimum) se trouve parmi les éléments les plus grands (resp. les plus petits) de chaque paire (et un éventuel élément supplémentaire si n est impair). On applique alors l'algorithme immédiat de recherche de maximum (resp. minimum) sur ces seuls éléments.

4. Combien de comparaisons (en fonction de n) sont-elles effectuées dans le pire cas par cet algorithme ?

$\lfloor n/2 \rfloor$ comparaisons sont nécessaires pour former les paires et $\lceil n/2 \rceil - 1$ pour les recherches de maximum et de minimum qui s'ensuivent, soit au total $\lceil 3n/2 \rceil - 2$ comparaisons.

On souhaite maintenant montrer que l'algorithme précédent est *optimal* dans le modèle et avec la métrique considérée. Pour cela on imagine un *adversaire* qui peut choisir le résultat des comparaisons effectuées, et qui cherche à garantir qu'il faudra toujours en effectuer au moins 'un certain nombre. Plus précisément, soit $\mathbb{A}$ un algorithme résolvant correctement le problème, et lors de toute exécution, chaque fois que l'algorithme compare deux éléments qui n'ont pas encore été comparés c'est l'adversaire qui choisit le résultat, de façon à maximiser le nombre de comparaisons effectuées.
Lors d'une exécution de $\mathbb{A}$, on note $\mathcal{N}$ (pour « nouveau ») l'ensemble des éléments n'ayant été comparé à aucun autre, $\mathcal{G}$ (pour « grand ») l'ensemble des éléments ayant été comparé à au moins un autre élément, et ayant toujours été comparé comme « plus grand », $\mathcal{P}$ (pour « petit ») l'ensemble des éléments ayant été comparé à au moins un autre élément, et ayant toujours été comparé comme « plus petit », $\mathcal{M}$ (pour « moyen ») l'ensemble des éléments ayant été comparé à au moins deux autres éléments, dont au moins l'un est plus petit et l'autre plus grand. On note le nombre d'éléments de chacun de ces ensembles (dans l'ordre) par le quadruplet (i, j, k, ℓ).

5. Que vaut $i + j + j + \ell$, à tout point de l'exécution de $\mathbb{A}$?

n.

6. Que vaut (i, j, k, ℓ) au début de l'exécution, et à la fin d'une exécution *toujours correcte* ?

Au début de l'exécution le quadruplet vaut $(n, 0, 0, 0)$, et il doit valoir $(0, 1, 1, n-2)$ à la fin d'une exécution toujours correcte (il ne faut notamment plus avoir aucun élément dans $\mathcal{N}$, qui peut sinon toujours contenir le vrai maximum ou minimum (ce qu'un adversaire ne se privera pas de décider)).

7. Développez une stratégie pour l'adversaire qui démontre l'optimalité de votre second algorithme.

Le but de l'adversaire est de montrer que pour tout algorithme correct $\mathbb{A}$, il existe des exécutions nécessitant au moins $\lceil 3n/2 \rceil - 2$ comparaisons pour déterminer le maximum et le minimum. Ces exécutions correspondent aux cas où l'algorithme « progresse lentement » quoi qu'il arrive, du fait du résultat des comparaisons (choisies par l'adversaire).

Pour garantir ce progrès relativement lent, on observe qu'un élément qui se trouve dans $\mathcal{M}$ n'a plus jamais besoin d'être comparé, et l'on peut donc chercher à remplir $\mathcal{M}$ le plus lentement possible. Les situations amenant un élément à être transféré dans $\mathcal{M}$ sont les suivantes :

- comparaison de deux éléments de $\mathcal{G}$ entre eux (quelque soit le résultat) : un élément transféré dans $\mathcal{M}$ depuis $\mathcal{G}$ (ℓ augmente de 1, j diminue de 1) ;
- comparaison de deux éléments de $\mathcal{P}$ entre eux (quelque soit le résultat) : un élément transféré dans $\mathcal{M}$ depuis $\mathcal{P}$ (ℓ augmente de 1, k diminue de 1) ;
- comparaison d'un élément x de $\mathcal{G}$ et d'un élément y de $\mathcal{P}$ *si* $x < y$: les deux éléments sont transférés dans $\mathcal{M}$ (ℓ augmente de 2, j et k diminuent de 1) ;
- comparaison d'un élément x de $\mathcal{N}$ avec un élément y de $\mathcal{G}$ *si* $x > y$: y transféré dans $\mathcal{M}$ depuis $\mathcal{G}$, x transféré dans $\mathcal{G}$ depuis $\mathcal{N}$ (ℓ augmente de 1, i diminue de 1) ;
- comparaison d'un élément x de $\mathcal{N}$ avec un élément y de $\mathcal{P}$ *si* $x < y$: y transféré dans $\mathcal{M}$ depuis $\mathcal{P}$, x transféré dans $\mathcal{P}$ depuis $\mathcal{N}$ (ℓ augmente de 1, i diminue de 1).

On peut alors remarquer que dans les deux derniers cas, si l'élément de $\mathcal{N}$ est comparé plus *petit* (resp. plus *grand*) à l'élément de $\mathcal{G}$ (resp. $\mathcal{P}$), aucun élément ne sera transféré à $\mathcal{M}$. De même, il n'y a aucun transfert à $\mathcal{M}$ dans le troisième cas si l'élément de $\mathcal{G}$ est comparé plus grand à l'élément de $\mathcal{P}$. Il s'ensuit que l'adversaire peut choisir le résultat des comparaisons de sorte que les seuls transferts vers $\mathcal{M}$ possibles le sont par des comparaisons entre éléments de $\mathcal{G}$ ou de $\mathcal{P}$ entre eux, et que celles-ci augmentent ℓ de 1.
Si l'on analyse maintenant le nombre de comparaisons nécessaires (pour cette stratégie de l'adversaire) pour atteindre un quadruplet $(0,1,1,n-2)$ depuis un quadruplet $(n,0,0,0)$, il faut :

- au moins $\lceil n/2 \rceil$ comparaisons avec des éléments de $\mathcal{N}$, nécessaires pour vider ce dernier ensemble (tout élément de $\mathcal{N}$ est transféré vers $\mathcal{G}$ ou $\mathcal{P}$ à sa première comparaison) ; par ce qui précède et avec la stratégie suivie, aucune de ces comparaisons ne change $\mathcal{M}$;
- au moins $n-2$ comparaisons pour remplir $\mathcal{M}$.

Soit au total au moins $\lceil n/2 \rceil + n - 2 = \lceil 3n/2 \rceil - 2$ comparaisons.

N.B. Cet exercice pourrait éventuellement laisser croire qu'il est assez aisé de prouver des *bornes inférieures* pour le coût des algorithmes, et de là prouver que des algorithmes sont optimaux. Il n'en est rien : réussir à prouver des bornes inférieures est plus l'exception que la règle en informatique.

Exercice 3. *Qu'est-ce qu'un algorithme ?*
Les énoncés des exercices précédents demandaient de décrire *informellement* les algorithmes, mais que serait une description *formelle* ? Pour vous, qu'est-ce qu'un algorithme ? En quoi un algorithme se différencie-t'il d'un programme ? Qu'est-ce qui différencie l'algorithmique de la programmation ?

REMARQUE : Les réponses à ces questions sont loin de faire consensus.